Cache Write Policies and Performance

Norman P. Jouppi
Digital Equipment Corporation Western Research Lab
250 University Avenue
Palo Alto, CA 94301

Abstract

This paper investigates issues involving writes and caches. First, tradeoffs on writes that miss in the cache are investigated. In particular, whether the missed cache block is fetched on a write miss, whether the missed cache block is allocated in the cache, and whether the cache line is written before hit or miss is known are considered. Depending on the combination of these polices chosen, the entire cache miss rate can vary by a factor of two on some applications. The combination of no-fetch-on-write and write-allocate can provide better performance than cache line allocation instructions. Second, tradeoffs between write-through and write-back caching when writes hit in a cache are considered. A mixture of these two alternatives, called *write caching* is proposed. *Write caching* places a small fully-associative cache behind a write-through cache. A write cache can eliminate almost as much write traffic as a write-back cache.

1. Introduction

Most of the extensive literature on caches has concentrated on read issues (e.g., miss rates when treating stores as reads), or writes in the context of multiprocessor cache consistency. However, uniprocessor[1] write issues are in many ways more complicated than read issues, since writes require additional work beyond that for a cache hit (e.g., writing the data back to the memory system).

The cache write policies investigated in this paper fall into two broad categories: write hit policies, and write miss policies.

Unlike instruction fetches and data loads, where reducing latency is the prime goal, the primary goal for writes that hit in the cache is reducing the bandwidth requirements (i.e., write traffic). This is especially important if the cycle time of the CPU is faster than that of the interface to the second-level cache, and if multiple instruction issue allows store traffic approaching one per cycle to be sustained in many applications. The write traffic into the second-level cache primarily depends on whether the first-level cache is *write-through* (also called *store-through*) or *write-back* (also called *store-in* or *copy-back*). Write-back caches take advantage of the temporal and spatial locality of writes (and reads) to reduce the write traffic leaving the cache.

Write miss policies, although they do affect bandwidth, focus foremost on latency. Write miss policies include three semi-dependent variables. First, writes that miss in the cache may or may not have a line allocated in the cache (*write-allocate* vs. *no-write-allocate*). If a cache uses a no-write-allocate policy, when reads occur to recently written data, they must wait for the data to be fetched back from a lower level in the memory hierarchy. Second, writes that miss in the cache may or may not fetch the block being written (*fetch-on-write* vs. *no-fetch-on-write*). A cache that uses a fetch-on-write policy must wait for a missed cache line to be fetched from a lower level of the memory hierarchy, while a cache using no-fetch-on-write can proceed immediately. We emphasize that write-allocate and fetch-on-write are not synonymous as commonly assumed. This paper investigates the combination of write-allocate but no-fetch-on-write which has superior performance over other policies. A new third variable of write policy, *write-before-hit*, is also investigated in this paper. If writes use the same pipeline timing as reads to reduce structural hazards in the pipeline, writes will occur before hit or miss is known. Obviously write-before-hit is only useful with write-through caches; if used with a write-back cache unique dirty data will be overwritten. Writes using write-before-hit that miss in the cache may simply invalidate cache lines "erroneously" written and pass the data written on to lower levels in the memory hierarchy. Different combinations of these three write-miss policy variables can result in a 2:1 range in cache miss rates for some applications.

Out of the hundreds of papers on caches in the last 15 years [14, 15], Smith [12] was the only paper to exclusively deal with write issues. This paper discussed write buffer

[1]By uniprocessor we include non-coherency issues in multiprocessor cache memories, as well as uniprocessor cache memories.

0884-7495/93 $3.00

performance for write-through caches, but did not investigate merging of pending writes to the same cache line by a write buffer. Smith [13] and Goodman [6] both have a section on write-back versus write-through caching, but they study only mixed first-level caches with traces under a million references. Among the more recent work in uniprocessor cache issues, Agarwal [1] and Hill [7] assumed write references were identical to read references in their analysis. Przybylski [10] includes write overheads in his analysis, but only considers the case of write-back caches at all levels. Write miss policies have been even less investigated. Almost all of the known results in the literature have been for the combination of write-allocate and fetch-on-write. The VAX 11/780 [2] and 8800 [3] were notable exceptions to this and used no-write-allocate. An unpublished paper by Smith [16] has a section that considers tradeoffs between write-allocate and no-write-allocate. It uses traces up to ten million references, but investigates only 4KB data caches with 16B lines.

Section 2 briefly describes the simulation environment and benchmarks used in this study. Policies for write misses, specifically fetch-on-write, write-allocate, and write-before-hit are investigated in Section 3. Section 4 investigates write hit tradeoffs between write-back and write-through caching, as well as ways of reducing write-through traffic. Section 5 summarizes the results of the paper.

2. Experimental Environment

This paper investigates write policies in the context of a modern memory hierarchy. One or more levels of on-chip caching are assumed, although the data in the paper is for the effects of these policies on the first-level cache performance. Because one or more levels of on-chip caching are assumed, the first-level cache sizes studied are from 1KB to 128KB, which are suitable for implementation on a VLSI chip.

Separate instruction and data caches are assumed at the first level, since these are necessary for superscalar and other types of high performance machine design. Only direct-mapped first-level data caches are studied.

The results in this paper were obtained by modifying a simulator for the MultiTitan [8] architecture. The MultiTitan architecture does not support byte loads and stores, so byte writes appear as word read-modify-writes. However, the number of byte operations in the programs studied are insignificant, so this does not significantly affect the results presented. Each experiment involved simulating the benchmarks, and not analyzing trace tapes.

The characteristics of the test programs used in this study are given in Table 2-1. Although six is a small number of benchmarks, the programs chosen are quite diverse, with two numeric programs, two CAD tools, and two Unix utilities. However, operating system execution, transaction-processing code, commercial workloads (e.g., COBOL), and multiprocessing were beyond the scope of this study. The benchmarks used are reasonably long in comparison with most traces in use today.

program name	dynamic instr.	data reads	data writes	total refs.	program type
ccom	31.5M	8.3M	5.7M	45.5M	C compiler
grr	134.2M	42.1M	17.1M	193.4M	PC board CAD
yacc	51.0M	12.9M	3.8M	67.7M	Unix utility
met	99.4M	36.4M	13.8M	149.7M	PC board CAD
linpack	144.8M	28.1M	12.1M	185.5M	100x100
livermore	23.6M	5.0M	2.3M	31.0M	loops 1-14
total	484.5M	132.8M	54.8M	672.8M	

Table 2-1: Test program characteristics

3. Write Misses: Fetch-on-Write, Write-Allocate, and Write-Before-Hit

The policy used on a write that misses in the cache (i.e., "write miss") can significantly affect the total amount of cache refill traffic, as well as the amount of time spent waiting during cache misses. The number of cache misses due to writes varies dramatically depending on the benchmark used. Figure 3-1 shows the percentage of misses that are due to writes for various cache sizes with 16B lines, using write-allocate with fetch-on-write. Figure 3-2 shows the percentage of misses that are due to writes for an 8KB cache with various line sizes. On average over all the cache configurations, write misses account for about one-third of all cache misses. Since loads outnumber stores in these benchmarks by roughly 2.4:1 (see Table 2-1), this means that stores are about as likely to cause a miss as loads.

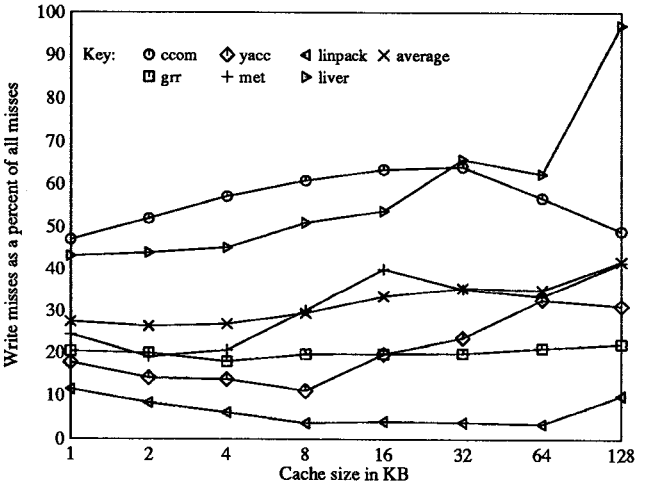

Figure 3-1: Write misses vs. cache size for 16B lines

There are four combinations of three write-miss policies from which to choose (see Figure 3-3).

In systems implementing a *fetch-on-write* policy, on a write miss the line containing the write address is fetched. In systems implementing a *write-allocate* policy, the address written to by the write miss is allocated in the cache. Note that it is possible to have a write-allocate policy without using fetch-on-write: here the data being written is writ-

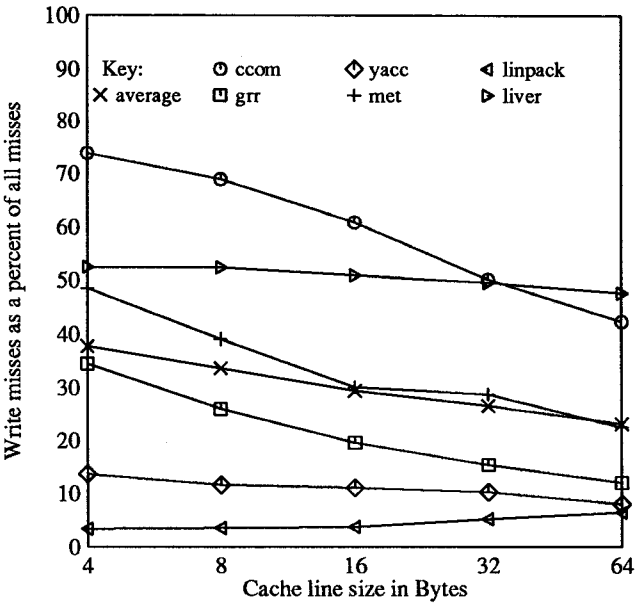

Figure 3-2: Write misses vs. line size for 8KB caches

Fetch-on-write?

Write-allocate?	Fetch-on-write	Write-before-hit?	
Yes	Yes	Write-validate	No
	Yes	Write-validate	Yes
No	No	Write-around	No
	No	Write-invalidate	Yes

Figure 3-3: Write miss alternatives

ten into the cache without fetching the old contents of the cache line. As compared to systems with non-blocking writes [17] which just write into a write buffer on a write miss, this combination can give significant performance improvement because the data written into the cache can be read later without a cache miss.

If a direct-mapped write-through cache is being used, the data can be written concurrently with the tag check. We call this a *write-before-hit* policy. If the tag does not match, the data portion of the line has been corrupted (i.e., assuming the line size is larger than the amount of data being written, the data is a mixture of information from two cache lines). If the system is fetch-on-write or write-allocate, the normal miss processing will restore the cache line to a consistent state. However, if the system is no-fetch-on-write and no-write-allocate, the line can simply be marked invalid, since the data is being written to a lower level in the memory hierarchy anyway. This invalidation can usually be done in a single cycle, or sometimes even in parallel with subsequent cache accesses, and so it is much faster than fetching the correct contents of the cache line being written.

A combination of fetch-on-write and no-write-allocate policies is not useful, since the old data at the write miss address is fetched but is discarded instead of being written into the cache. When referring to caching policies, fetch-on-write has been used to imply write-allocate in the literature. A fetch-on-write policy has the same result whether or not write-before-hit is used. If the old data at the write miss address is not fetched (i.e., no-fetch-on-write), three distinct options are possible. We call the combination of no-fetch-on-write and write-allocate *write-validate*. With write-validate, the line containing the write is not fetched. The data is written into a cache line with valid bits turned off for all but the data which is being written. For write-validate polices with write-before-hit, the valid bits for the old data must be turned off with an additional write operation once miss is detected. We call the combination of no-fetch-on-write, no-write-allocate, and no-write-before-hit *write-around*, since write misses do not go into the cache but go around it to the next lower level in the memory hierarchy, leaving the old contents of the line in place. Note that writes that hit in the cache still write into the cache with a write-around policy; here we are only considering write miss alternatives. The combination of write-before-hit, no-fetch-on-write, and no-write-allocate we call write-invalidate, because the line must be invalidated on a miss. Note that this is not the same as the *write-purge* of [16] which invalidates the cache line on a write hit. As defined here, write-invalidate only invalidates lines when it misses. Thus, write hits still write into the cache as usual with a write-invalidate write miss policy.

We call write misses that that do not result in any data being fetched with a write-validate, write-around, or write-invalidate policy *eliminated misses*. For example, with write-validate if the invalid part of a line is never read, the fetch of the data (and the attendant stalling of the processor) is eliminated. Only if the invalid portion of a line resulting from the write-validate strategy is read without first being written or the line being replaced, is this counted as a miss. Similarly, with write-invalidate only if the line being written or the old contents of the cache line are read before another address mapping to the same cache line misses is it counted as a miss. Finally, with write-around, only if the data being written is read before any other data which maps to the same cache line is read is the miss counted. This terminology neglects the time required to set the valid bits on an eliminated miss. However, if maintenance of the valid bits cannot be done in parallel with other operations, it typically takes at most a cycle, which is insignificant compared to cache miss penalties.

The write miss policy used is sometimes dependent on the write hit policy chosen. Write-around and write-invalidate (i.e., policies with no-write-allocate) are only useful with write-through caches, since writes are not entered into the cache. Fetch-on-write and write-validate can be used with either write-through or write-back caching.

Write-validate requires the addition of valid bits within a cache line. Valid bits could be added on a word basis, so that words can be written and the remainder of the line marked invalid. In systems that allow byte writes or unaligned word writes, byte valid bits would be required for a pure write-validate strategy. However, the addition of byte valid bits is a significant overhead (one bit per byte, or 12.5%) in comparison to a valid bit per word (3.1%). Thus, in practice machines with byte writes that have write-validate capability for aligned word and double-word writes would probably provide fetch-on-write for byte writes. Write-validate also requires that lower levels in the memory system support writes of partial cache lines.

In multiprocessor systems with cache consistency, write misses require traffic to gain exclusive write ownership of the block being written. In a multiprocessor using write-validate with an ownership protocol, a fetch with ownership still needs to be sent to the coherency point. When the fetched data returns it can be merged with the cache line which has been allocated based on the word valid bits. Thus extra coherency transactions required by write-validate can negate its traffic advantages over fetch-on-write in multiprocessor configurations. However, to the extent that the processor can continue execution with relaxed consistency models, including use of data which has been previously written but for which the rest of the line has not returned, improved performance would still be possible with write-validate as compared to a simple fetch-on-write policy.

The choice of write miss policy can make a significant difference in the performance of certain operations. For example, consider copying a block of information. If fetch-on-write is used, each write of the destination must hit in the cache. In other words, the original contents of the target of the copy will be fetched even though they are never used and are only overwritten with write data. This will reduce the bandwidth of the copy by wasting fetch bandwidth. Given a total bandwidth available for reads and writes, a fetch-on-write strategy would have only two-thirds of the performance on large block copies as a no-fetch-on-write policy since half of the items fetched would be discarded.

Some architectures have added instructions to allocate a cache line in cases where programmer directives specify or the compiler can guarantee that the entire cache line will be written and the old contents of the corresponding memory locations will not be read [11, 8, 4]. These instructions are limited to situations where new data spaces are being allocated, such as a new activation record on a process stack, or a new output buffer is obtained from the operating system. Unfortunately there are a number of problems that prevent broader application of software cache line allocation:

1. The entire cache line must be known to be written at compile time, or if some of the line is not written its old contents must not need to be saved. (In contrast, write-validate can allow partial lines to written, and is not subject to optimization limitations such as incomplete alias information, etc.)
2. Cache line sizes vary from implementation to implementation, limiting object code using these instructions to the machines with cache line sizes equal to or smaller than that assumed in the allocate instructions.
3. Context switches after a line has been allocated and partially written but before it has been completely written result in dirty and incorrect cache lines. (One way around this would be to add valid bits to each write quantum in the line, but this provides the hardware support needed for write-validate).
4. There is extra instruction execution overhead for the cache allocation instructions, or extra opcode space is used if they are merged with store instructions.

Thus, the use of cache line allocation instructions is limited to situations such as new data allocation and buffer copies. Write-validate can provide better performance than cache line allocation instructions since it is also applicable in cases where only part of a line is being written or it is not possible to guarantee that an entire line is written at compile time. Write-validate works for machines with various line sizes, and does not add instruction execution overhead to the program. Finally, since write-validate has word valid bits there are no problems with cache lines being left in an incorrect state on context switches.

A technique that has recently become popular in the literature is non-blocking (also called lock-up-free) caches [17]. Typically a cache with non-blocking writes implements what we have called a write-around policy. Some practical implementations of non-blocking writes may need to use write-invalidate, however, for timing reasons.

Figure 3-4 shows the reduction in write misses for write-validate, write-around, and write-invalidate for caches with 16B lines. We define the reduction in *misses as a percentage of write misses* (M%WM) for a policy X as follows:

$$M\%WM_{reduction} = 100 \times \frac{Misses_{fetch-on-write} - Misses_X}{WriteMisses_{fetch-on-write}}$$

Note that we take the difference in *total* misses (both read and write) and divide it by the number of *write* misses of fetch-on-write. When policies other than fetch-on-write are used this takes into account extra read misses that occur as a result of not using fetch-on-write. Since fetch-on-write fetches a cache line on every write miss, it corresponds to the X axis (0% reduction) in Figure 3-4. In general write-validate performs the best, averaging more than a 90% reduction in misses as a percentage of write misses. The two no-write-allocate strategies, write-around and write-invalidate, have an average reduction in misses as a percentage of write misses of 40-65% and 30-50% respectively. Write-around has a greater than 100% reduction in misses as a percentage of write misses for 32KB and 64KB caches when running *liver, because it saves read misses as well. liver* is a synthetic benchmark made from a series of loop kernels, and the results of loop kernels are not read by

successive kernels. However, successive loop kernels read the original matrices again. The range of cache sizes from 32KB to 64KB is big enough to hold the initial inputs, but not the results too. Since write-around does not place the results in the cache but keeps the old contents of the cache line unchanged, it can also result in fewer read misses since the initial data is not replaced with write data or invalidated.

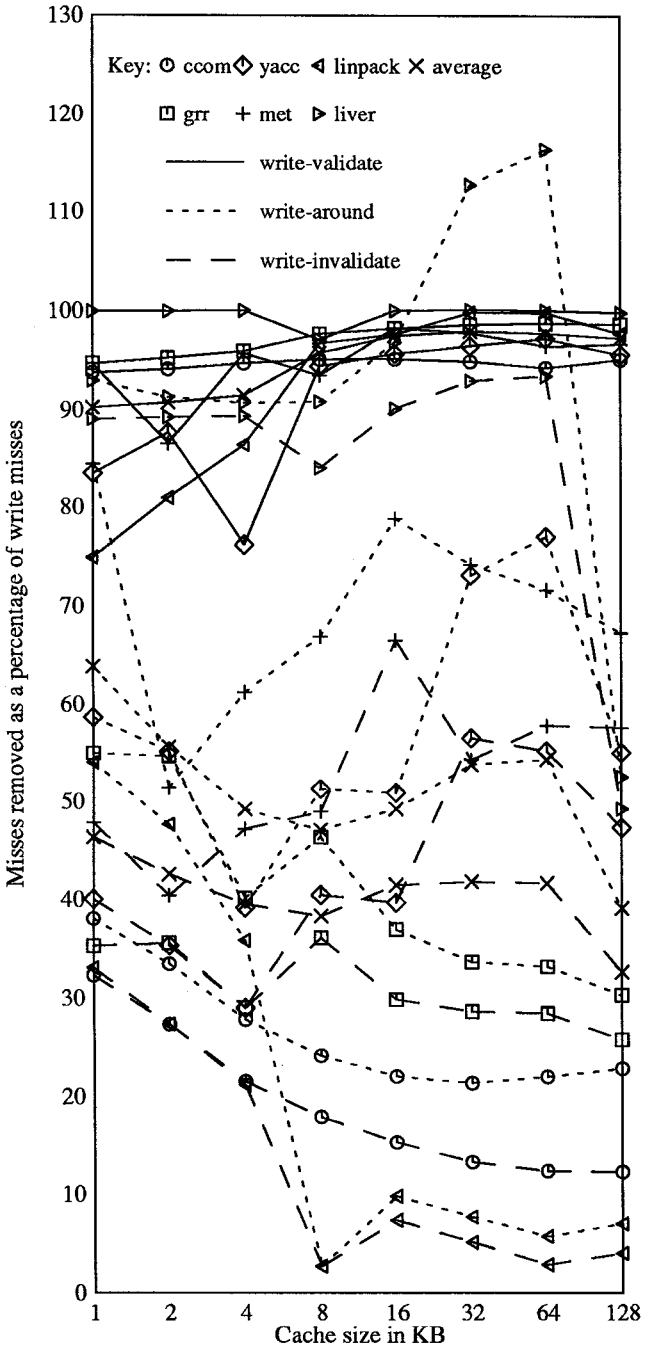

Figure 3-4: Miss reduction as % of write misses, 16B lines

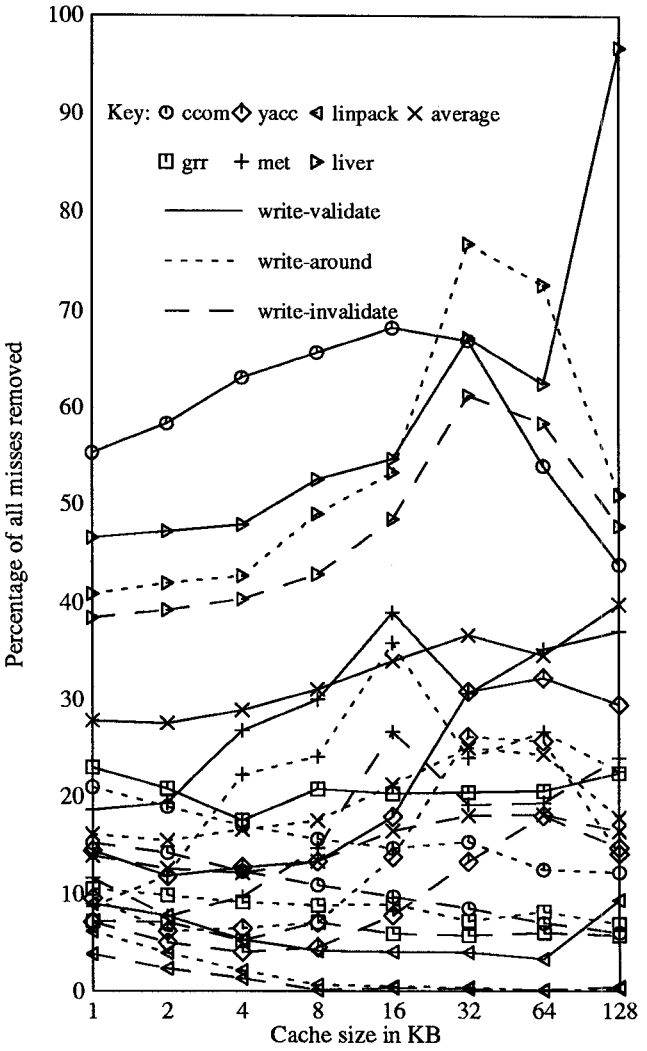

Figure 3-5: Total miss rate reductions for 16B lines

Figure 3-5 shows the reduction in data cache misses (including both read and write) for write-validate, write-around, and write-invalidate for caches with 16B lines. The overall reduction in miss rate (MOverall) is computed as folows:

$$MOverall_{reduction} = 100 \times \frac{Misses_{fetch-on-write} - Misses_X}{Misses_{fetch-on-write}}$$

Figure 3-5 is basically Figure 3-4 multiplied by Figure 3-1. *ccom* and *liver* benefit the most from a write-validate policy. This can be explained as follows. Many of the operations in *ccom* and *liver* are similar to copies: data is read but other data is written. For example, array operations of the form "for j := 1 to 1000 do A[j] := B[j] + C[j]" only write data which is never read before being written. Similarly, write-validate would be useful for a compiler if it has a number of sequential passes, each one reading the data structure written by the last pass and writing a dif-

ferent one. The other programs have more read-modify-write behavior. The best example of this is *linpack*. The inner loop of *linpack*, saxpy, loads a matrix row and adds to it another row multiplied by a scalar. The result of this computation is placed into the old row. Here write-validate would be of very little benefit since almost all writes are preceded by reads of the data anyway. On average over the six programs write-validate reduced the total number of data cache misses (over both read and write) by 31% for an 8KB data cache with 16B line size.

Write-around performs well when the data being written by the processor is not read by it soon or ever. This is the situation in *liver* with a 32KB or 64KB cache, the only benchmark that performs better with write-around than write-validate. In general, however, most programs are more likely to read what they have just written than they are to re-read the old contents of a cache line. For all other cases the performance of write-around is worse than that of write-validate.

Write-invalidate does not show as much improvement over fetch-on-write as the other two strategies, but it still performs surprisingly well. *livermore* has about a 40% reduction in misses, and the six benchmarks on average have a 10-20% total reduction in misses compared to fetch-on-write. Moreover, write-invalidate is very simple to implement. In a write-through cache using write-invalidate the data can be written at the same time the tags are probed. If the access misses, the line has been corrupted so it can be simply marked invalid, often without inserting any machine stall cycles.

Figure 3-6 shows the reduction in misses as a percentage of write misses for write-validate, write-around, and write-invalidate for 8KB caches with various line sizes. Since fetch-on-write fetches a cache line on every write miss, it corresponds to the X axis (0% reduction) in Figure 3-6. Write-validate, write-around, and write-invalidate have the highest benefit for small lines. If the line size is the same as the item being written, any old data fetched by fetch-on-write is merely discarded when the write occurs. As the line size gets larger, the odds that some old data on the line will be needed increases, so the advantage of write-validate decreases. The miss rate reduction of write-around also decreases with increasing line size for similar reasons. The performance advantage of write-invalidate decreases with increasing line sizes because more information is being thrown away. Again write-validate performs the best, averaging more than a 90% reduction in misses as a percentage of write misses except at the longest line sizes. The two no-write-allocate strategies, write-around and write-invalidate, have an average reduction in misses as a percentage of write misses of 40-70% and 35-50% respectively.

Figure 3-7 shows the overall reduction in total misses for write-validate, write-around, and write-invalidate for 8KB caches. (This graph is basically Figure 3-6 multiplied by Figure 3-2.) Again write-around generally performs worse

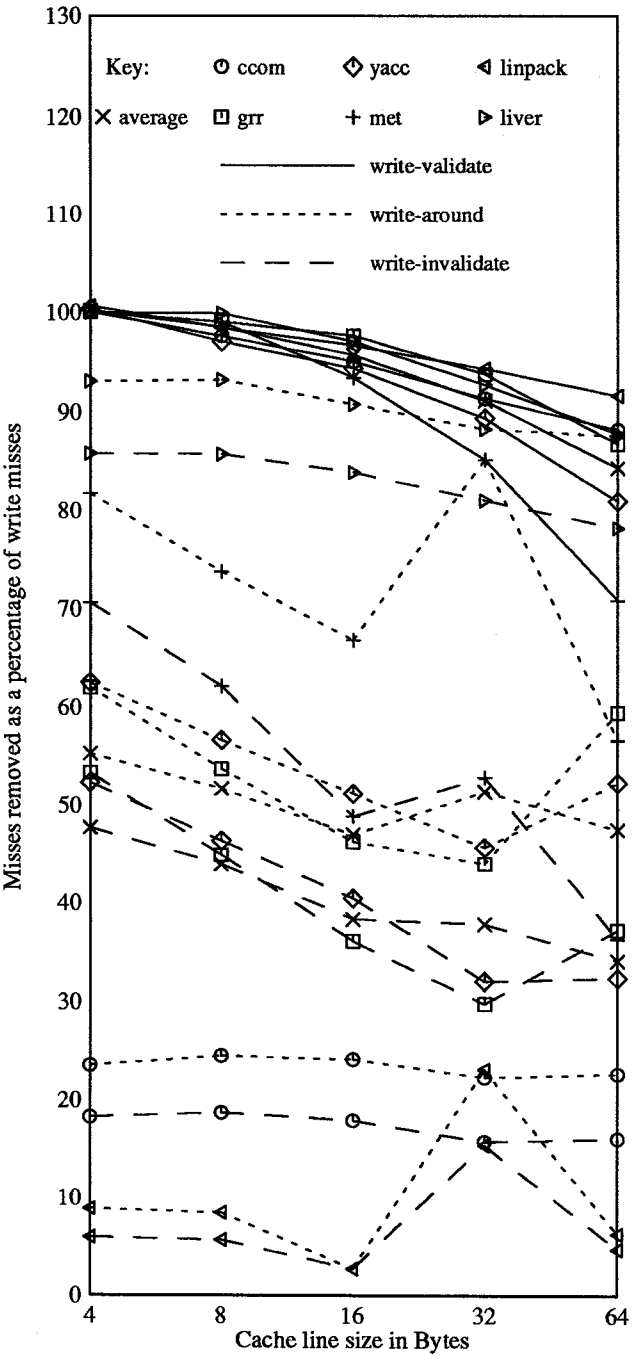

Figure 3-6: Miss reduction as % of write misses, 8KB caches

than write-validate, because most programs are more likely to read the data that was just written than the old contents of the cache line. Both write-validate and write-around perform better than write-invalidate, but again write-invalidate performs surprisingly well.

We can generate a partial order of the relative total read and write miss traffic between these four write-miss policy combinations (see Figure 3-8). Fetch-on-write always has the most lines fetched, since it fetches a line on every miss.

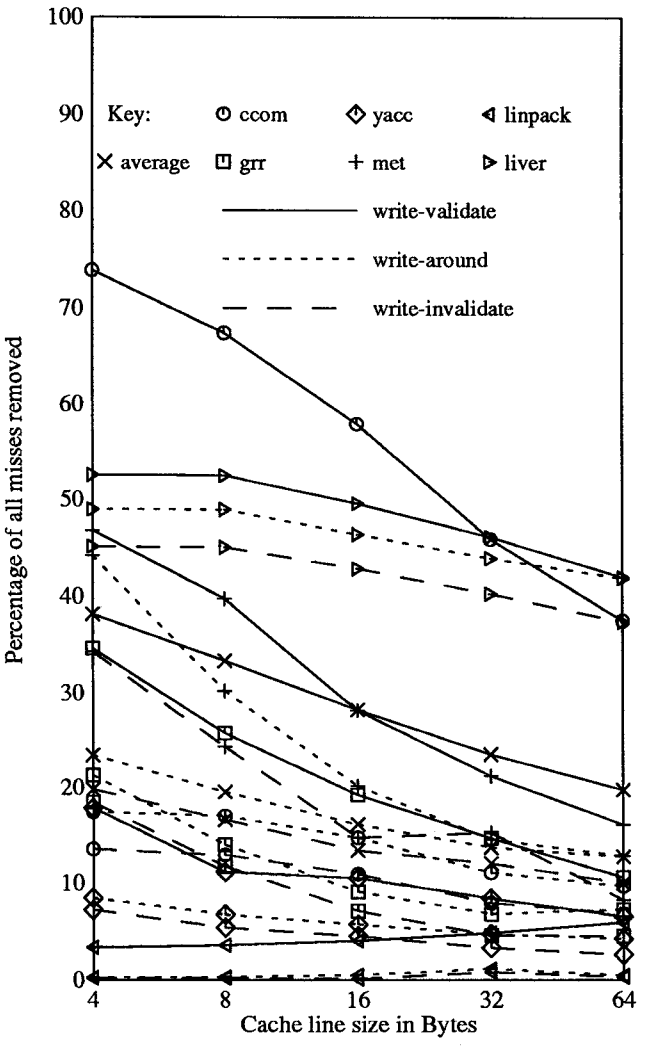

Figure 3-7: Total miss rate reduction for 8KB caches

Write-invalidate avoids misses in the case where neither the line containing the data being written nor the old contents of the cache line are read before some other line mapping to the same location in the cache is read. This saves some misses over fetch-on-write. Write-around and write-validate always have fewer misses than write-invalidate. Write-around avoids fetching data in the same cases as write-invalidate, as well as cases where the old contents of the cache line are accessed next. Write-validate avoids fetching data in the same cases as write-invalidate, as well as cases where the data just written is accessed next. Usually the data just written (i.e., write-validate) is more useful than the old contents of the cache line (i.e., write-around), but this is not always the case. Also, the ratio of miss rate reduction of write-validate to write-around decreases as the line size increases since write-validate invalidates an increasing number of bytes while write-around leaves all the bytes on the line valid.

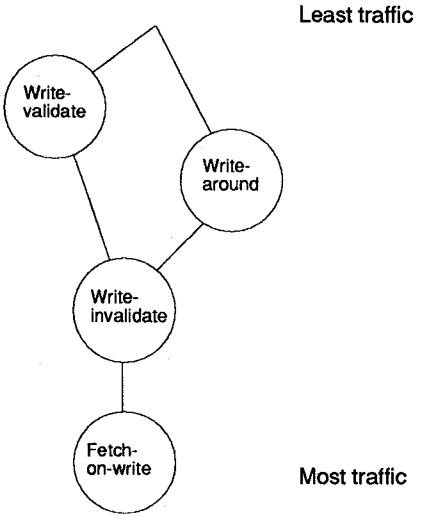

Figure 3-8: Relative order of total fetch traffic

4. Write Hits: Write-Through vs. Write-Back

When a write hits in a cache, two possible policy choices exist. First, the data can be written both into the cache and passed on to the next lower level in the memory hierarchy. This policy is called write-through. A second possible policy on write hits is to only write the data to the first-level cache. Only when a dirty line (i.e., a line that has been written to) is replaced in the cache is the data transferred to a lower level in the memory hierarchy. This policy is called write-back. Write-back caching takes advantage of the locality of reference of writes to reduce the amount of write traffic going to the next lower level in the memory hierarchy.

Although the conventional wisdom may be that write-back caching is always preferred over write-through caching, in multilevel cache hierarchies there are a number of significant advantages of write-through caching for first-level caches. In the common case where the first-level caches are on-chip, the second-level cache is typically write-back.

One advantage of write-through caching is the write bandwidth into the cache (i.e., the number of cycles required per write). A write-back cache must probe the tag store for a hit before the corresponding data is written. This is because if the write access misses and the victim is dirty, unique dirty data will be lost if the cache line is written before the probe. However, a direct-mapped write-through cache can always write a cache line of data at the same time as probing the address tag for a hit. If the access misses, the line is never dirty and will be replaced anyway so there is no problem. If the data cache is set-associative, the probe must occur before the write whether the cache is write-back or write-through. However, a large and increasing number of first-level data caches are direct-mapped, for reasons discussed in [7, 10]. The two-cycle access of straightforward write-back and set-associative cache im-

plementations (i.e., a probe cycle followed by a write cycle) provides more limited store bandwidth at the input to the cache than a direct-mapped write-through cache, in order to reduce the write bandwidth required on the output side of the cache. In machines that can issue multiple instructions per cycle, the incoming load/store bandwidth of the cache can be a limiting factor to machine performance. Although stores are about half as frequent as loads on average, if each store requires two cycles this will result in a 33% reduction in effective first-level cache bandwidth as compared to a machine that only requires one cycle per store. There are also more complicated methods for reducing write-hits to a single cycle in write-back and set-associative write-through caches [5] which are beyond the scope of this work.

A second advantage of write-through caching over write-back caching is the ease with which stores and their attendant writes are integrated into the machine pipeline (see Figure 4-1). In a direct-mapped write-through cache writes can always be performed in the pipestage where loads read the cache. If the access turns out to be a miss the conventional miss-recovery hardware provided for load misses can be used, and the store write cycle is simply repeated. However, a simple write-back or set-associative write-through cache can require two cycles of cache access per store: the first cycle probes the cache tags, and the second sets the appropriate dirty bits and writes the data. This will require interlocks when loads immediately follow stores, since the stores would be accessing the data section at the same time as the next (load) instruction is accessing the data section of the cache (i.e., without interlocks the WB pipestage of the store would be at the same time as the MEM pipestage of the load.) Note that if load latency weren't important, loads could delay their data access until WB after hit or miss were already known. Then stores and loads would access the cache with the same timing and could be issued one per cycle in any order. However, since load latency is of critical importance in machine design, this is not a viable option. Although in Figure 4-1 stores into a write-through cache would commit a pipestage earlier than loads or other operations (which commit in WB), the cache line written by the store can be flushed a pipestage after its write without adverse consequences. This allows exceptions to be handled precisely. Similarly, data going into the write buffer in the MEM pipestage of Figure 4-1 can be aged one cycle until the instruction is known to have completed without exception.

A third advantage of write-through caching over write-back caching is error-tolerance, for both manufacturing or hard defects and soft defects. A write-through cache can function with either hard or soft single-bit errors, if parity is provided. This is because the write-through cache contains no unique dirty data, and reads of data with errors can be turned into cache misses. A write-back cache can not tolerate a single-bit error of any type unless ECC is provided. ECC must usually be computed on at least a 32

		store timing	
	load	write-	write-
pipestage	timing	through$	back*
instr fetch			
register fetch			
address calc.			
cache	read data	write data	read tags
access	read tags	read tags	
write			write data
registers			if tags hit

$ Also assumes direct-mapped.
* Also applies if set-associative write-through.

Figure 4-1: Direct-mapped write pipelines

bit data word to be economical. For example, single bit detection and correction (but not double detection) ECC requires 6 bits per 32 bit word versus 4 bits per 8 bit byte giving 16 bits per 4 bytes. Thus operations like byte store must first read and ECC-decode a word before being able to write a byte. Moreover, byte parity on a four-byte word would allow four single-bit errors to be corrected by refetching a write-through line in comparison to only one error for an ECC-protected write-back cache word. This is true even though byte parity requires only two-thirds of the overhead of word ECC. Thus write-through caches with parity have better error-tolerance at a smaller cost than write-back caches with ECC.

The primary problem with write-through caches is their higher write traffic as compared to write-back caches. One way to reduce this traffic is to use a *coalescing write buffer*, where writes to addresses already in the write buffer are combined.

Figure 4-2 shows the simulation results for an 8-entry coalescing write buffer. Each write buffer entry is a cache line (16B) wide. The data presented are the results of the six benchmarks averaged together. Simulations were performed where the write buffer emptied out an entry every n cycles, with n varying from 0 to 48 cycles. In practice the number of cycles between retirement of write buffer entries will depend on intervening cache miss service and other system factors. However, as on-chip processor cycle times become much faster than the off-chip cycle times, and processors issue more than one instruction per cycle, the number of off-chip cycles between writes can become very large (e.g., more than 12). Since cache miss service effectively stops processor execution in many processors, cache misses were ignored in Figure 4-2. This allows a fixed time between writes to be used as a reasonable model of the write buffer operation. If dirty write buffer entries are written back quickly, they do not stay in the write buffer for many cycles and hence relatively little merging takes place. For example, if write buffer entries are retired every 5 cycles, the write traffic is reduced by only 10%. The only way that a significant number of writes are merged (e.g., 50% or more) is if the write buffer is almost always full. But in this case stores almost always stall because no write

buffer entries are available. For example, to attain a write traffic reduction of 50%, writes must be retired no more frequently than every 38 cycles, resulting in a CPI burden of 7! Since much of current computer research is focused on achieving machines with CPIs of less than one, write buffer stalls should be well under 0.1 CPI. This means that only a small percentage of writes (e.g., less than 20%) can be merged with simple coalescing write buffers. The extra traffic resulting from this lack of coalescing wastes cache bandwidth that could otherwise be used for prefetching or other uses.

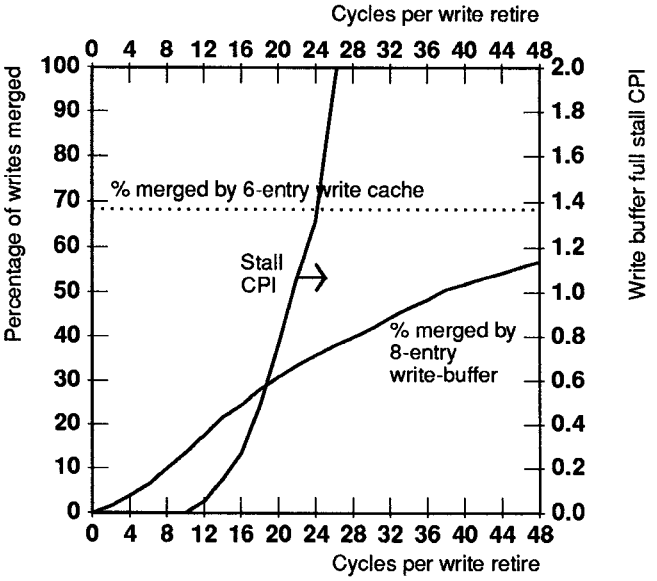

Figure 4-2: Coalescing write buffer merges vs. CPI

Instead of having writes enter and leave the write buffer as soon as possible, we can add a *write cache* in front of the write buffer and behind the data cache. A write cache is a small fully-associative cache (see Figure 4-3). With a small number of entries we can try to coalesce the majority of writes and decrease the write traffic exiting the chip. When a write misses in the write cache, the LRU entry is transferred to the write buffer to make room for the current write. In actual implementation, the write cache can be merged with a coalescing write buffer. Here a write buffer of m entries would only empty an entry if it has more than n valid entries, where n is the number of entries conceptually in the write cache (with $m > n$). A write cache can also be implemented with the additional functionality of a victim cache [9], in which case not all entries in the small fully-associative cache would be dirty. Note that whereas a first-level data cache which can probed and written in one cycle is very difficult to achieve (for an interesting cycle time), a several-entry fully-associative write-cache could be easily implemented within a machine cycle. This is because each tag in the write cache must has its own comparator. Thus there is no tag RAM access time before the tag comparisons can begin, as would be the case in a ordinary data cache.

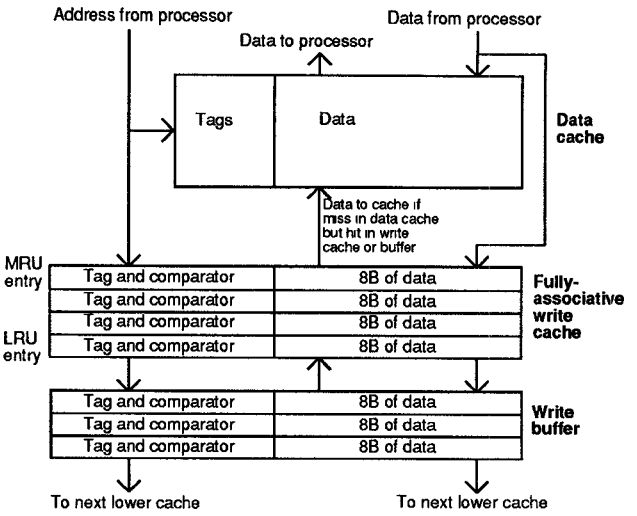

Figure 4-3: Write cache organization

Figure 4-4 gives the number of writes removed by a write cache with varying numbers of 8B lines. (8B was chosen as the write cache line size since no writes larger than 8B exist in most architectures, and write paths leaving chips are often 8B.) A write cache of only five 8B lines can eliminate 50% of the writes for most programs. Two notable exceptions to this are *linpack* and *liver*. Because these programs sequentially travel through large arrays, even a write-back cache of modest size (less than 32KB) removes very few writes. In order to get a better idea of how write caches compare with write-back caches, the write traffic reduction of a write cache is given relative to a 4KB write-back cache in Figure 4-5. In Figure 4-5 a write cache of only four 8B entries removes over 50% of the writes removed by a 4KB write-back cache on all of the benchmarks except *met*. Another interesting result is that a write cache with eight or more 8B entries actually outperforms a 4KB direct-mapped write-back cache on *liver*. This is because mapping conflicts within the write reference stream prevent a direct-mapped write-back cache from being as effective at removing write traffic as the fully-associative write cache.

Figures 4-4 and 4-5 also give the average traffic reduction of write caches in absolute terms and relative to a write-back cache. The two most interesting points on these curves are probably a five-entry write cache, since it seems to be at the knee of the traffic reduction curve, and a one-entry write cache, since it is the simplest to implement. The five-entry write cache can remove 40% of all writes, or 63% of those removed by a 4KB write-back cache. The single-entry write cache can remove 16% of all writes on average, which is 21% of the writes removed by a write-back cache.

Of course the relative traffic reduction of a write cache varies as the size of the write-back cache used in the comparison varies (see Figure 4-6). Compared to a 1KB write-back cache, a five-entry write cache removes 72% of the

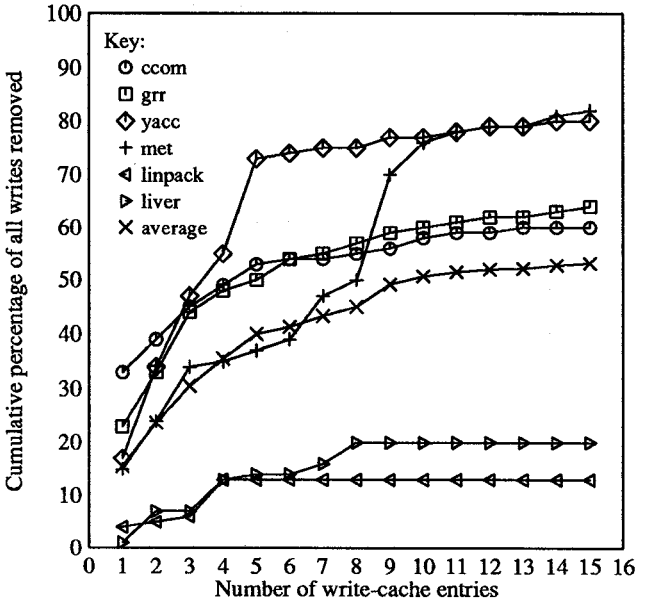

Figure 4-4: Write cache absolute traffic reduction

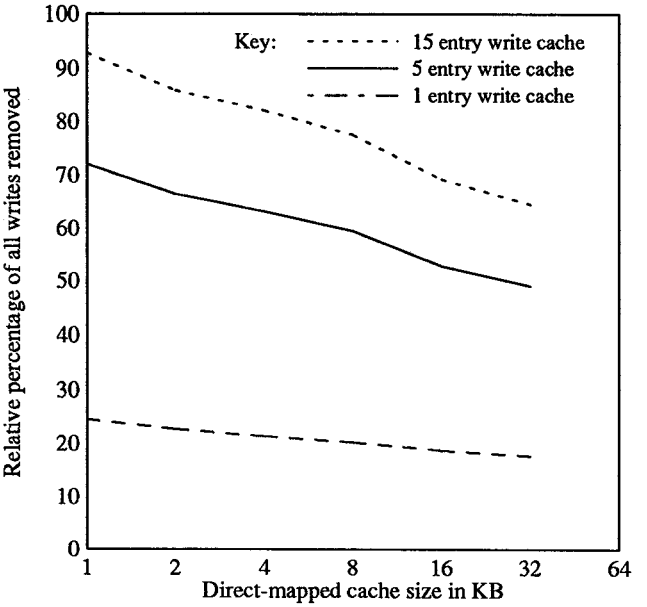

Figure 4-6: Write cache traffic reduction vs. cache size

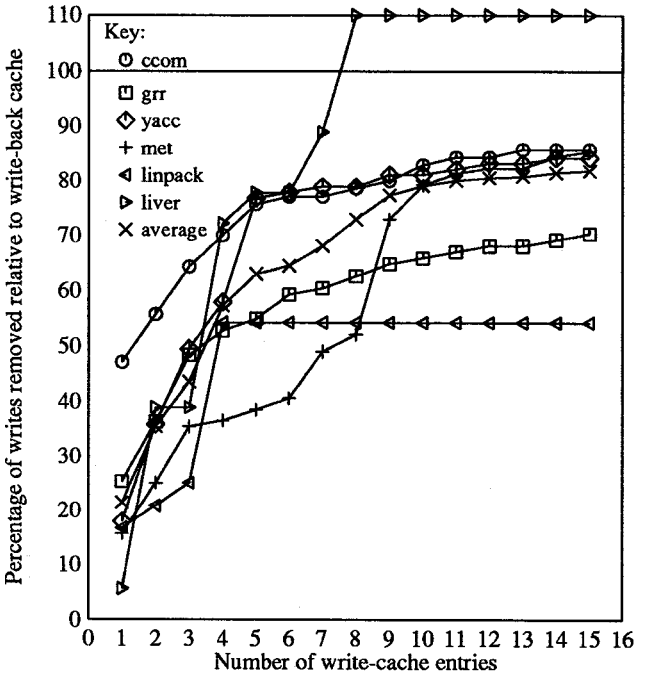

Figure 4-5: Write cache traffic relative to 4KB write-back

write traffic but compared to a 32KB write-back cache it only removes 49% of the write traffic. This change is surprisingly small considering the 32:1 ratio in write-back cache size, and is due to the write cache's good absolute traffic reduction. The reduction in write cache relative effectiveness is fairly uniform as the write-back cache size used for comparison increases in size.

5. Conclusions

An important performance issue involving writes is the policy for handling write data on a write miss. Four options exist: either fetch the line before writing (i.e., fetch-on-write), allocate a cache line and write the data while turning off valid bits for the remainder of the line (i.e., write-validate), just write the data into the next lower level of the memory hierarchy leaving the old contexts of the cache line intact (i.e., write-around), or invalidate the cache line and pass the data on to the next lower level in the memory hierarchy (i.e., write-invalidate). Write-invalidate is useful when writes occur in a direct-mapped write-through cache before hit or miss is known, and the line is corrupted on a miss. Of course if a write hits in the cache, the cache is written into as usual independent of the write miss policy. Systems with lock-up-free caches in the literature typically provide a write-around policy, although write-invalidate may need to be implemented in some machines due to timing constraints. Write-validate and write-around always outperform fetch-on-write. In general write-validate outperforms write-around since data just written is more likely to be accessed soon again than data read previously. Write-invalidate always performs worse than write-validate or write-around, but always outperforms fetch-on-write. For systems with caches in the range of 8KB to 128KB with 16B lines, write validate reduced the total number of misses by 30 to 35% on average over the six benchmarks studied as compared to fetch-on-write, write-around reduced the total number of misses by 15 to 25%, and write-invalidate reduced the total number of misses by 10 to 20%. Unlike cache line allocation instructions, write-validate is applicable to all write operations.

Moreover, it does not require compiler analysis or program directives, works with various line sizes, does not add any instruction execution overhead, and through the use of word valid bits allows a consistent and correct view of memory to be maintained.

An important write policy issue for writes that hit in a cache is write-through versus write-back caching. *Write caching*, a technique for reducing the traffic of write-through caches, was studied. It was found that a small fully-associative write cache of five 8B entries could remove 40% of the write traffic on average. This compares favorably to the 58% reduction obtained by a 4KB write-back cache. Since write-through caches have the advantage of only requiring parity for fault tolerance and recovery, while write-back caches require ECC, write-through caches seem preferable for small and moderate sized on-chip caches. Only when cache sizes reach 32KB does the additional traffic reduction provided by write-back caches over write-through caches become significant.

Acknowledgments

The author would like to thank the referees for their helpful comments, and those at DECWRL who had helpful comments on early drafts of this paper.

References

1. Agarwal, Anant. *Analysis of Cache Performance for Operating Systems and Multiprogramming*. Ph.D. Th., Stanford University, 1987.

2. Clark, Douglas W. "Cache Performance in the VAX 11/780". *ACM Transactions on Computer Systems 1*, 1 (February 1983), 24-37.

3. Clark, Douglas W., Bannon, Peter J., and Keller, James B. Measuring VAX 8800 Performance with a Histogram Hardware Monitor. The 15th Annual Symposium on Computer Architecture, IEEE Computer Society Press, June, 1988, pp. 176-185.

4. DeLano, Eric, Walker, Will, Yetter, Jeff, and Forsyth, Mark. A High-Speed Superscalar PA-RISC Processor. Compcon Spring, IEEE Computer Society Press, February, 1992, pp. 116-121.

5. Fu, John, Keller, James B., and Haduch, Kenneth J. "Aspects of the VAX 8800 C Box Design". *Digital Technical Journal 1*, 6 (February 1987), 41-51.

6. Goodman, James R. Using Cache Memory to Reduce Processor-Memory Traffic . The 10th Annual Symposium on Computer Architecture, IEEE Computer Society Press, June, 1983, pp. 124-131.

7. Hill, Mark D. *Aspects of Cache Memory and Instruction Buffer Performance*. Ph.D. Th., University of California, Berkeley, 1987.

8. Jouppi, Norman P. Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU. The 16th Annual Symposium on Computer Architecture, IEEE Computer Society Press, May, 1989, pp. 281-289.

9. Jouppi, Norman P. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers . The 17th Annual Symposium on Computer Architecture, IEEE Computer Society Press, May, 1990, pp. 364-373.

10. Przybylski, S.A. *Cache Design: A Performance-Directed Approach*. Morgan-Kaufmann, San Mateo, CA, 1990.

11. Radin, George. The 801 Minicomputer. (The First) Symposium on Architectural Support for Programming Languages and Operating Systems, IEEE Computer Society Press, March, 1982, pp. 39-47.

12. Smith, Alan J. "Characterizing the Storage Process and Its Effect on the Update of Main Memory by Write-Through". *Journal of the ACM 26*, 1 (January 1979), 6-27.

13. Smith, Alan J. "Cache Memories". *Computing Surveys 14*, 3 (September 1982), 473-530.

14. Smith, Alan J. "Bibliography and Readings on CPU Cache Memories". *Computer Architecture News 14*, 1 (January 1986), 22-42.

15. Smith, Alan J. "Second Bibliography on Cache Memories". *Computer Architecture News 19*, 4 (June 1991), 154-182.

16. Smith, Alan J. CPU Cache Memories. unpublished, draft of April 24, 1984.

17. Sohi, Gurindar, and Franklin, Manoj. High-Bandwidth Data Memory Systems for Superscalar Processors. Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, IEEE Computer Society Press, April, 1991, pp. 53-62.